Code 582

Flight Software Branch

C++ CODING STANDARD

Flight Software Branch – Code 582

Version 1.0 – 12/11/03

582-2003-004

Goddard Space Flight Center

Greenbelt, Maryland

National Aeronautics and
Space Administration

FORWARD AND UPDATE HISTORY

This standard defines the NASA Goddard Space Flight Center Flight Software Branch coding standards for flight software written in the C++ source language.

Version	Date	Description	Affected Pages
Draft	09/0503	Initial Document Draft to Goddard Space Flight Center (GSFC) Code 582 for review, developed by the Hammers Company.	all
Draft 1	09/19/03	Incorporated GSFC comments and delivered to GSFC Code 582	all
0.1	10/0303	Formatting & layout changes; new cover page, no change to technical content.	all
1.0	12/11/03	Incorporation of changes negotiated with Glenn Cammarata.	many

CONTENTS

1 INTRODUCTION AND OVERVIEW

1.1 Identification of Document

This document defines the C++ coding standards and style guidelines to be used for developing new C++ embedded software for the NASA Goddard Space Flight Center Flight Software Branch. The C++ language is defined in International Standard ISO/IEC 14882 by the National Committee for Information Technology Standards (NCITS).

This document is a selective collection of coding style and standards excerpts from numerous white papers and publications. It attempts to define a minimum set of style guidelines and coding standards that are helpful for embedded software development and are practical to implement.

The coding standards presented here are specifically tailored to working in an embedded environment, where the robustness and reliability of the software is critical. It is unacceptable for embedded software to periodically reboot because of exhausted or squandered resources.

1.2 Purpose and Objectives of Document

This document sets the policy for the appropriate use of C++ language constructs, indentation, brace placement, commenting, variable declaration, variable naming, and white space usage. In some cases specific use of valid C++ language constructs are restricted to prevent common logical and syntactical errors.

There are two primary goals for these coding standards and guidelines:

- Make the C++ code **readable**, so that it can be easily understood by other programmers in the Branch

- Make the C++ language **easier to use**, to facilitate development and maintenance of new projects

Ease of understanding, development and maintainability take priority over cleverly written or highly concise code.

Coding Standards are reserved for coding practices which bear directly on the reliability of code. Violating a precept of a code standard puts your code in jeopardy of error. The standards documented here are targeted at reducing errors (both compile-time and run-time) while developing code.

Style guidelines deal with issues such as formatting and naming conventions, and although it can be highly subjective, more than anything style affects the readability of your code. The establishment of a common and consistent style for a project will facilitate understanding and maintenance of code developed by more than one programmer, and will make it easier for several people to cooperate in the development of the same program.

Discipline is necessary if coding style and standards are to be strictly followed, and the more difficult or tedious an organization's style and standards are, the less likely that the developers will faithfully follow the procedures. This document attempts to maintain a balance between flexibility and thoroughness.

1.3 How to Use This Document

The standards described in this document are either mandatory or discretionary. Mandatory standards are requirements that must be followed. These are indicated by the word **shall**. Discretionary standards are guidelines that allow some judgment or personal choice by the programmer. These standards are indicated by the use of the word **should**. A programmer should have a good reason when they choose to disregard a discretionary standard. Where possible, this document gives examples of acceptable deviations. A mandatory standard can be waived by the Development Team Lead (DTL) for specific cases, where appropriate. The DTL shall document the standards waived in an addendum, including a rationale for each waiver. Existing code is not bound by these standards. However, any modifications for either maintenance or re-use should comply with these standards or the standards documented by the original project.

For the standards to be effective, strict enforcement is required. Standards should not be violated. If an exception to a standard is identified as necessary for a specific piece of code in a project, a waiver should be sought from the DTL and the exception (and waiver) clearly noted in the code. This documentation provides crucial information to other developers when maintaining the code and it forces the original programmer to consider and justify why the standard does not apply.

If the exception is not a single case, but is applicable across a given project, then a project-specific addendum should be created as described above.

The primary mode of reading code is assumed to be on a computer screen, with an editor that provides syntax colorization and code navigation. There will be a few times, primarily at code reviews, when code will be printed out. This document reflects this bias towards on-screen viewing.

Coding standards are often thought of as a nuisance and a restriction by many programmers. The exact opposite is actually the case. Standards relieve the programmer from wasting time on the mundane parts of coding, allowing the programmer to focus on software design. In a team environment standards provide uniformity in programming style which aids communication. Standards provide a legend into everyone’s code so anytime one needs to discuss a piece of code with another team member no time is wasted on getting oriented. Code reviews can focus on what the code is doing and the content of the comments. Coding standards make the code easier to read, understand, and to maintain. Writing code that is easily readable and works takes more time than simply writing code that works. However, the time saved in the maintenance phase, and in re-use for other projects, more than makes up for the time spent writing. This document explicitly favors ease of reading over ease of writing.

2 REFERENCE DOCUMENTS

The following documents were used as source material during the development of this coding standard and style guide:

Document or Link	Release Date	Source
Flight Software Branch C Coding Standard	7 Sept 2000	GSFC Flight Software Branch
C++ Coding Standards	28 Feb 2002	Mike Blau/GSFC
Swift-BAT C++ Coding Standards	25-June-2002	GSFC, SWIFT-BAT Project
SECCHI Flight Software Development and Coding Standards	19-Dec-2001	the Hammers Company, Inc.
The Embedded C++ Programming Guide Lines Version WP-GU-003	Copyright(C) 6-Jan-1998	Embedded C++ Technical Committee
The C Programming Language, Second Edition	1988	Brian Kernighan and Dennis Ritchie
www.research.att.com/~bs/bs_faq2.html	2003	Bjarne Stroustrup
International Standard ISO/IEC 14882, Programming Language - C++ http://www.ncits.org/cplusplus.htm	1998	NCITS (National Committee for Information Technology Standards)

3 STANDARDS AND STYLE GUIDE

3.1 General Rules

3.1.1 Compiler options shall be selected such that non-conformances to the ISO/IEC 14882 standard are elevated to errors.

3.1.2 A given software unit shall only reference data applicable to its unit. If a unit references data from multiple units, then a redesign may be in order.

3.1.3 Explicit constants, (i.e., "hard coded" numbers) should not be used, except to establish identifiers to represent them (0 and 1 may be exceptions).

3.1.4 Use of preprocessor directives or macros should be minimized in C++. The exception to this is using the `#ifndef` read-once latch header file definition, as described in this document.

3.1.5 Only one object should be declared per declaration:

Not Acceptable

```
int *x, y;    // pointer to int and int in same declaration
```

Acceptable

```
int* x;       // declaration for pointer to int
int  y;       // declaration for int
```

3.1.6 Variables shall not be declared within loops, as this may cause performance issues; variables should be declared within the proper scope (with the exception of loops, as noted); if a variable is only needed within a sub-block of a function or method, declare it within that sub-block. Locals are better than class variables, which are better than subsystem globals, which are better than system globals.

Not Acceptable

```
while( true )    // variable declared within loop
{
    int c;
    // do something with c
}
```

Acceptable

```
int c;           // variable declared outside of loop

while( true )
{
    // do something with c
}
```

3.1.7 `iostream` should not be used, because the streams library is large and slow. An alternative is to overload the `<<` operator.

3.1.8 `asserts` shall not be included in delivered code; they should only be enabled during unit testing. Do not use asserts to trap bad ground commands, user errors, etc.; these should be handled with system events and command error counters. Do not use asserts with Interrupt Service Routines (ISRs).

3.1.9 Auto-generated code, e.g., UML tools, MATLAB, should not be modified. If manual changes are required and code is regenerated, care must be taken to preserve the manual changes.

3.1.10 Types based on `int8, int16, int32, int64, float32, float64, uint8, uint16, uint32, uint64`, etc., should be `typedef`'d instead of relying on the default action of the compiler, for example, when using `int, long, float, short` etc.

3.1.11 A `const` shall never be converted to a non-`const`. If a non-`const` is really needed, the code should be restructured.

3.1.12 Type casting should not be used; however, if necessary, use C++ style type-casts, i.e., `static_cast, cont_cast, dynamic_cast, reinterpret_cast`. The C++ style type casts offer protection that the standard C style type-casts do not.

3.1.13 `static` declarations inside methods should be used sparingly, taking into account any reentrance issues that may result.

3.1.14 The C++ constructs `true` and `false` shall be used for Boolean values.

3.1.15 `throw/catch` should not be used. An exception may be granted if the compiler supports thread-safe exception handling.

3.1.17 `throw/catch` shall not be used in Interrupt Service Routines (ISRs).

3.2 Dynamic Memory Allocation

Dynamic allocation is generally not used. If the decision is made to allow Dynamic Memory Allocation on a specific project, this section applies.

3.2.1 Objects should be allocated only once at initialization, immediately following boot-up.

In the event that objects must be allocated after initialization, then the following apply:

3.2.2 Memory shall be de-allocated in the same scope in which it was allocated.

3.2.3 `malloc` and `free` shall not be used. In C++, memory is allocated and de-allocated with `new` and `delete`, respectively.

3.2.4 Variables or objects created with `new` shall be deleted with `delete`.

3.2.5 Array delete shall be used for array objects, as shown in the following examples:

```
ptr = new type;
// do something with ptr;
delete ptr;
```

- or -

```
ptr = new type[ count ];
// do something with ptr;
delete [ ]ptr;
```

3.2.6 A pointer shall never be used after it has been deleted.

3.3 Files

3.3.1 File extensions should be *.cpp and *.h or *.hpp. The base filename should match between the header and implementation files.

3.3.2 Base filename should indicate what class is defined therein. For example, cProjectClass should be defined in cProjectClass.hpp and implemented in cProjectClass.cpp.

3.3.3 Each file should define or implement one class. It is sometimes acceptable to combine closely related classes in one file (e.g., table class with its corresponding table item class).

3.3.4 The grouping of methods, within each section (public, private, protected), should be consistent throughout all classes for a project. The order can be alphabetical, by return type, or functionality (ground command accessor). The order that member methods are defined in the header file should match the order in which they appear in the implementation file. This allows the header file to provide a look and feel for the layout of the implementation file.

3.3.5 File length, including white space and comments, should not exceed 1000 lines.

3.3.6 Header files shall be protected with an `#ifndef` read-once latch. The first two lines in every header file are:

```
#ifndef THISFILENAME_H_
#define THISFILENAME_H_
```

The last line in every header file is:

```
#endif    // end of THISFILENAME
```

3.3.7 Each header file shall `#include` the files it needs to compile, rather than forcing users to `#include` the needed files. `#includes` shall be limited to what the header needs; other `#includes` should be placed in the source file.

3.3.8 Header files shall not generate object code.

3.3.9 Quotes should be used to include files in the project source tree; <> should be used for files in the compiler or OS trees.

3.4 Classes

3.4.1 A class should be declared with a maximum of one of each of the following sections, in the order shown:

```
1. public
2. protected
3. private
```

3.4.2 Depending on the class, there are certain methods that should be implemented:

Constructor: A constructor shall be explicitly defined. The constructor's argument use, if any, should be clearly commented.

Destructor: The destructor shall be virtual if the class is intended to be inherited by other classes. Virtual destructors ensure objects will be completely destructed regardless of inheritance depth.

Copy Constructor: A copy constructor and assignment operator shall be defined if the class objects are intended to be copied. If the class objects are not intended to be copied, the copy constructor and assignment operator shall be private, with no bodies defined for them. Assume that class objects will NOT be copied until the copy operations are needed.

Assignment Operator: An assignment operator shall be defined if the class objects are intended to be assigned. If the class objects are not intended to be assigned, then the assignment operator shall be private, with no body defined. Assume that class objects will NOT be assigned unless assignment operations are needed.

The rationale here is that the compiler will implicitly generate default versions if these are not explicitly defined. The best practice is to disallow the compiler to generate default code. Where possible, these items should be explicitly defined when their use is anticipated.

3.4.3 The public scope of every class should be minimized. Public data members shall never be used; all data members should be private. If needed, members may be elevated to protected. Data members shall be accessed by appropriate data access methods, e.g,:

```
SetMyVariable( type value );          //sets data member to value
type MyValue = GetMyVariable( void ); //returns value of MyValue
```

Note: Refer to 3.6.2 in regard to methods that accept no parameters.

3.4.4 `friend` classes shall never be used.

3.4.5 Multiple inheritance shall never be used.

3.4.6 The use of global variables and functions should be avoided. All variables and functions should be members of a class (class members and methods, respectively).

3.4.7 Classes should be used instead of structures.

3.4.8 A class's declaration should make clear the logical usage of the class. Implementation details should be hidden.

3.4.9 Upon creation, an object created from a class shall be in a valid state, ready to be used.

3.4.10 User-defined types shall be passed by reference. Unless the intent of the method is to modify an argument, the argument in the method signature should be declared as a reference to a const object. Note the following example, in which X is a user-defined type:

Not Acceptable

```
A MyClass::MethodA( X  MyArg  );     // object passed by value
B MyClass::MethodB( X* pMyArg );     // object passed by pointer
```

Acceptable

```
C MyClass::MethodC( const X&  MyArg ); // MyArg NOT to be
                                       // modified
D MyClass::MethodD(       X&  MyArg ); // MyArg to be modified
```

3.4.11 Code shall never return a reference or pointer to a local variable.

3.5 Class Methods - Inline

3.5.1 Inline functions shall only be used for access methods (`Get/Set`), and limited to three (3) lines or less. Caution must be exercised if the class contains a large number of access methods. A large number of inline methods may cause resource or compatibility issues.

3.5.2 Inline methods definitions shall be placed in the implementation (body) file.

3.6 Class Methods – Non Inline

3.6.1 The return type of a method shall be provided explicitly.

3.6.2 Methods that accept no parameters should have an explicit parameter list of `void` (except for constructors and destructors). Although not required by the C++ language, explicitly listing `void` avoids any confusion regarding the parameters.

3.6.3 Class methods should use const wherever possible, but not on a method that modifies data outside of the class in which it resides.

3.6.4 The `virtual` keyword shall be used to identify a modified inherited method:

```
class MyBaseClass
{
    virtual void MyFunc();
}
```

The following code ensures that `MyDerivedClass::MyFunc` overrides `MyBaseClass::MyFunc`

```
class MyDerivedClass : MyBaseClass
{
    virtual void MyFunc();
}
```

3.6.5 The formal arguments of methods shall have names, and the same names should be used in both the method declaration and the method definition.

3.6.6 When declaring methods, the leading parenthesis and the first argument (if any) should be written on the same line as the method name. If space permits, other arguments and the closing parenthesis may also be written on the same line as the method name. Otherwise, each additional argument is to be written on a separate line (with the closing parenthesis directly after the last argument).

3.6.7 Method length, including white space and comments, should not exceed 100 lines.

3.7 Constructors and Destructors

3.7.1 When possible, constructors should initialize each data member in the class to a valid default value. The object should be completely initialized.

3.7.2 Members should appear in an initialization list in the order in which they are declared, one per line and equally indented.

3.7.3 Constructors and Destructors shall never be inline.

3.7.4 `delete` should be called on all pointer members in a destructor.

3.7.5 Objects created with `new` should, when possible, be `delete`'d in the reverse order, i.e.,

```
A a = new A;
B b = new B;
// do something with a
// do something with b
delete b;
delete a;
```

3.8 Overloaded Operators

3.8.1 Overloaded Operators shall not change the meaning of C++ operators. For example, "+" should mean "add", etc.

3.8.2 When an operator has an opposite, (such as == and !=), both shall be defined.

3.8.3 If the operator "=" is overloaded, it shall return a reference to `*this`.

3.8.4 If the operator "="is overloaded, all data members shall be assigned.

3.8.5 If the operator "="is overloaded, it shall include an assignment to self.

3.9 Flow Control

3.9.1 The flow control primitives `if, else, while, for`, and `do` should be followed by a block, even if it is an empty block. At times, everything that is to be done in a loop may be easily written on one line in the loop statement itself. It may then be tempting to conclude the statement with a semicolon at the end of the line. This may lead to misunderstanding since, when reading the code it is easy to miss such a semicolon. Instead place an empty block after the statement to make completely clear what the code is doing. If an empty block is used, a comment indicating that the block is intentionally left blank should be used to indicate that the code is complete.

3.9.2 "Within tolerance" shall be used instead of testing for exact equality when testing equality on floating-point numbers.

Not Acceptable

```
if( someVar == 0.1 ) // may never be evaluated as true
```

Acceptable

```
if( abs(someVar - 0.1) < MAX_TOLERANCE )  // safe
```

The constant 0.1 cannot be represented exactly by any finite binary mantissa and exponent.

3.9.3 The use of the loop constructs `do, while`, and `for` shall follow the recommendations of Brian Kernighan and Dennis Ritchie (The C Programming Language, Second Edition, 1988). The `while` is used to control a loop until a condition becomes true. The `for` is used for simple initialization and increment operations. The `do` is used under the same conditions as the `while` except that the loop will always be executed first before testing the condition.

3.9.4 Nesting levels greater than six (6) deep should not be used. If more than 6 levels of nesting are required, consideration should be given to dividing the method or function into smaller ones.

3.9.5 The ternary conditional operator (`?:`) shall not be nested.

3.9.6 Only one exit point (`return`) should be used in a single method.

3.9.7 The code following a `case` label shall be terminated by a `break` statement. If several `case` statements execute the same block of code, i.e., fall-throughs, they must be clearly commented.

3.9.8 A `switch` statement shall always contain a `default` case which handles unexpected cases.

3.9.9 `breaks` shall only be used in `switch` statements, and nowhere else.

3.9.10 `goto` shall never be used.

3.9.11 `unsigned` shall be used only for variables that cannot have negative values. Variables representing size or length are typical candidates for `unsigned` declarations.

3.9.12 Inclusive lower limits and exclusive upper limits shall be used. Instead of saying that x is in the interval `x>=23` and `x<=42`, use the limits `x>=23` and `x<43`. The following important claims then apply:

1. The size of the interval between the limits is the difference between the limits.
2. The limits are equal if the interval is empty.
3. The upper limit is never less than the lower limit.

3.9.13 The `continue` statement shall never be used.

3.9.14 Logical operators shall not be used on non-Boolean values:

Not Acceptable

```
int x;
// code that modifies x
if( !x )
{
// do something
}
```

Acceptable

```
int x;
// code that modifies x
if( x==0 )
{
// do something
}
```

3.9.15 Parentheses shall be used to clarify the order of evaluation for operators in expressions; There are a number of common pitfalls having to do with the order of evaluation for operators in an expression. Binary operators in C++ have associativity (either leftwards or rightwards) and precedence. If an operator has leftwards associativity and occurs on both

sides of a variable in an expression, then the variable belongs to the same part of the expression as the operator on its left side. For example:

```
x = 7 + 18 % 11 % 2;    // expected result may differ from actual
```

With no parenthesis, the above expression will be evaluated as follows:

```
x = 7 + (( 18 % 11 ) % 2 );  // expected behavior made clear
```

and will result in a value of x = 8. However, depending on the desired order of evaluation, the expression can be written as:

```
x = ( 7 + 18 % 11 ) % 2;   // this now evaluates to zero
```

with a different result.

With the exception of the simplest expressions, parenthesis must always be used to clarify the order of evaluation.

3.9.16 C++ allows the overloading of operators, something which can easily become confusing. For example, the operators `<<` (shift left) and `>>` (shift right) are often used for input and output. Since these were originally bit operations, it is necessary that they have higher priority than relational operators. This means that parentheses shall be used when outputting the values of logical expressions. For example, if it is decided to use the streams library, and it is desired to output the sum of x and y:

Not Acceptable

```
cout <<   x + y;
```

Acceptable

```
cout << ( x + y );
```

3.9.17 Side effects within expressions should be used sparingly. It is recommended that no more than one operator with a side effect (`op =, ++, --`) appear within an expression. Function calls with side effects count as operators for this purpose. Note the following examples:

Not Acceptable

```
status = func( paramA,  ++paramB );
```

Acceptable

```
paramB++;
status = func( paramA,  paramB );
```

3.9.18 In an `if-else` chain, the form

```
if( condition )
{
    statement;
}
else if( condition )
{
    statement;
}
else if( condition )
{
    statement;
}
else
{
    statement;
}
```

should only be used when the conditions are all the same basic type (such as testing the same variable against different values), and the conditions involved are mutually exclusive.

If the conditions are qualitatively different, the additional if statements should start on new lines, indented, as in:

```
if( condition1 )
{
    statement;
}
else
{
    if( condition2 )
    {
        statement;
    }
    else
    {
        statement;
    }
}
```

3.9.19 An `if---else if---` chain shall always end with an `else` clause.

3.9.20 The `switch` statement should be used instead of long `if---else if---` chains whenever tests for equality of scalar expressions are being made. This is both more readable, and when compiled, produces more efficient code.

3.9.21 `||` and `&&` should not be used with right-hand operands having side effects.

3.9.22 Whenever `||` and `&&` are mixed in the same expression, parentheses should be used for clarity.

3.10 Code Formatting and Style

3.10.1 Styles for brackets and indenting shall be consistent throughout the project.

3.10.2 The format of the `switch` statement should observe the following guidelines:

1. A `switch` statement and every `case` in it (including `default`) should be preceded by a blank line when this improves readability (small `switch`es may ignore this rule). Multiple `case` labels on a single block of code should be on separate lines, but they should not be separated by blank lines.

2. The code after a `case` must be indented at least three spaces from the `switch` statement.

3. In general, a `switch` statement should exhibit the following form:

```
// some code;
switch (expression)
{
    case constant-expression 1:
        statement;
        ...
        statement;
        break;

    case constant-expression 2:
        statement;
        ...
        statement;
        break;

    default:
        break;
}
// some more code;
```

3.10.3 Braces delineating blocks shall be in one of the following styles:

```
(A)  if (exp)
     {
        code
     }


(B)  if (exp){
         code
     }


(C)  if (exp)
        {
            code
        }
```

The style chosen should remain consistent.

3.10.4 All code should be indented a minimum of three (3) spaces for each indentation level. The indention level must be consistent for all blocks in a file.

3.10.5 Statements at the same logical nesting level should be at the same level of indentation.

3.10.6 Tabular indentation shall not be used. Use space characters rather than embedded horizontal tab characters.

3.10.7 Each line should contain only one (1) statement.

3.10.8 Blank lines should be used between blocks of code that are functionally distinct (e.g. to separate large control blocks).

3.10.9 A minimum of two (2) blank lines should separate individual functions.

3.10.10 A minimum of one (1) blank line(s) should separate the declarations and the code in a method.

3.10.11 Blank spaces should be used anywhere where spacing improves readability.

3.10.12 Spaces should not be used around '.' or '->', or between unary operators and operands; Code is more readable if spaces are not used around the '.' or '->' operators. The same applies to unary operators (those that operate on one operand), since a space may give the impression that the unary operand is actually a binary operator.

3.10.13 Source lines shall not exceed 120 characters in length. We want to encourage long meaningful identifier names; this is easier to accommodate if lines can be longer than 80 characters.

3.10.14 Groups of data assignments should be aligned, as in the following example:

```
const int lngth   = 1000;
const char mychar = 'c';
const myval       = 12440;
```

3.11 Comments

3.11.1 All code shall be appropriately commented, especially if the code is highly complex.

3.11.2 A block comment with the appropriate configuration management tags should be included at the top of all files.

3.11.3 Code that has been "commented out" shall be removed before software is released.

3.11.4 All class header files shall contain a prolog. A standard prolog is best. The following information should be contained in the prolog:

- File Name

- Description
- Developer Name(s)
- Organization (i.e., GSFC, Code 582, etc.)
- Proprietary Notices
- Version Control (from Configuration Management Tool)

A project specific prolog that contains this information should be used for every header file. Header files shall also contain the read-once latch described in this document. The description in the implementation file should simply be “See Header”.

All methods should contain the following information. This provides visual separation for each method as well as basic information:

- Method Name
- Description
- Arguments
- Returns
- Information on how to call method
- Notes

3.11.5 Full English sentences or intelligible phrases shall be used for comments.

3.11.6 The purpose of code blocks should be explained with separate comments indented at the same level as the code.

3.11.7 Comments should be made clear and readable by including small blocks of comments as "paragraph headers" to a block of code rather than to comment every line or so, especially if the code is straightforward (which it should be).

3.11.8 A comment should follow the declaration of a variable whenever it clarifies its usage.

3.11.9 Statement and variable comments should line up.

3.11.10 If it is absolutely necessary to use global variables, each variable shall have a comment explaining its usage.

3.11.11 Anything obvious should not be commented, as this only serves to obscure relevant comments, the code itself, and the natural flow of the logic.

3.11.12 No comment should be followed by program code on the same line.

3.11.13 A space shall be inserted between the text of a comment and the delimiters `//`, `/*`, and `*/`.

3.12 Naming Conventions

3.12.1 Variable names should never begin with an underscore ("_"), as this may cause compiler issues.

3.12.2 Class names must identify the type of object they represent, as follows:

[Note: we don't have complete agreement on this paragraph. If you are a DTL using version 1.0 of this standard, any waivers you request to this paragraph will be considered as potential updates to the next version of this standard.]

Types. The name assigned to a type shall have two parts. The first part shall be a prefix that helps to identify what category of type the name is associated with. The prefix shall be a single lower case letter as defined in the table below. The second part of the type name shall be a sequence of words describing the usage or functionality encapsulated by the type. Each word in the name shall be capitalized and there should be no underscores between the words.

Type	Prefix	Example
Class	c	`cCcsdsPrimaryHeader`
Struct	s	`sAcsTelemetryData`
Union	u	`uUpDownHardwareReg`
Enum	e	`eCmdStatus`

Methods. The name assigned to a method shall have two parts. The first part shall be a prefix that identifies which high level software component owns the method. The prefix shall be as defined in the table below. The second part of the name shall be a sequence of words describing the action performed by the method. The first word in the name shall be lowercase with subsequent words capitalized. There should be no underscores between words. The first and second parts of the name are separated by an underscore.

Type of Method	Prefix	Example
System global method	SY_	`SY_copyToDownlink`
Subsystem global	XX_ (XX = subsystem code)	`SH_processNextSlot`
Class member method	*none*	`setPacketLength`

Variables. The name assigned to a variable shall have two parts. The first part shall be a prefix that identifies the context under which the variable was created and the high-level software component that owns it. The prefix shall be as defined in the table below. The second part of the name shall be a sequence of words describing the purpose of the variable. The first word in the name shall be lowercase with subsequent words capitalized. There should be no underscores between words. The first and second parts of the name are separated by an underscore.

Type of Variable	Prefix	Example
Local	*none*	`loopCounter`
System Global	sy_	`sy_systemClock`
Subsystem Global	xx_ (xx = subsystem code)	`to_filterTable`
Class Member	m_	`m_timeAtLast1Hz`
Structure Element	*none*	`myStructureElement`

Constants. The name assigned to a `const` variable or `#define` shall be all uppercase letters with words separated by underscores. The first two letters should indicate the subsystem defining the constant, e.g., `SM_MAX_PKT_SIZE`.